Towards Secure Erlang Systems

Alexandre Jorge Barbosa Rodrigues
Klarna Bank AB
Stockholm, Sweden
alexandre.rodrigues@klarna.com

Viktória Fördős
Klarna Bank AB
Stockholm, Sweden
viktoria.fordos@klarna.com

Abstract

At Klarna we handle customer's data with utmost care since we believe protecting data is one of the most basic obligations of any companies. Achieving our goal requires more effort since the Erlang ecosystem was designed for private networks where the concern about malicious users performing attacks was not given relevance. Therefore, every day we also put on our security hat to develop and review code changes going into production. In this paper, we show the challenges that today's Erlang systems are faced with and explain why *all* Erlang developers and operators must have a security aware mindset.

CCS Concepts • **Security and privacy** → *Distributed systems security*; *Software security engineering*;

Keywords Erlang system security, Erlang distribution protocol, TLS, software analysis

ACM Reference Format:
Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. 2018. Towards Secure Erlang Systems. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3239332.3242768

1 Introduction

Recent years have shaped the way we use Erlang. Erlang is now everywhere; it drives national health care, banks and online betting portals, just to give some examples. These are all mission-critical systems with high availability. In every minute thousands of users send requests towards these systems *through the internet*. Nonetheless, nobody has revisited the security design of Erlang, therefore, the following statement, that Joe Armstrong made in [1] still holds:

> Our idea was to connect stock hardware through TCP/IP sockets and run a cluster of machines behind a corporate firewall. We were not interested in security since we imagined all our computers running on a private network with no external access. This architecture led to a form of all-or-nothing security that makes distributed Erlang suitable for programming cluster applications running on a private network, but unsuitable for running distributed applications where various degrees of trust are involved.

This means that the Erlang language was designed assuming the network is secure but this is not how Erlang is used today. Erlang systems are not isolated, they communicate with the external world using various protocols, such as HTTP, TCP and XMPP. Hence, Erlang developers and operators must deal with the topic of security.

But how can one secure their Erlang system? Today there is no straightforward answer to this question.

Contribution This paper is a statement paper, a discussion initiator. The goal of the paper is to raise security awareness in the Erlang community. In this paper, we show security-related challenges in the Erlang world. We discuss techniques and tools that the community can use to improve the security of their Erlang systems.

2 Related Work

Erlang/OTP is a powerful general use platform that is shipped with libraries that allow developers to create systems that communicate with the external world. Notwithstanding, security of Erlang systems is a fairly unexplored area, still there have been some contributions done by the community.

Erlang's distribution protocol and the remote procedure call library were identified as 'insecure by default' since they rely on plain text communication, an easily breakable authentication mechanism and a port mapper daemon (*epmd*) that is vulnerable to forgery attacks [4].

Since then, proposals to have remote procedure call over the Secure Shell Connection protocol [5] and industry-driven implementation over TLS [1] have been made available. Improvements and bug fixes of the Erlang distribution over TLS and respective documentation have been done by the OTP team. Despite these relevant contributions, no major

[1] https://github.com/bet365/erpc

Erlang '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242768

re-design in ERTS level having security in mind was done yet.

These facts force the security aware part of the community to look at the problem from a different angle. Michael Truog, the author of the *Primitive Erlang Security Tool* (PEST)[2], approaches the problem through source code analysis to find usages of vulnerable functions. PEST can be used to check Erlang modules compiled with `debug_info` and warn the developer of, for instance, the usage of functions that may allow bash script injection if not protected with input validation (e.g., the `cmd` function of the `os` module) or the usage of the `binary_to_atom` or `list_to_atom` that may make the system crash when the number of atoms created reaches the pre-defined system limit (by default the maximum number of atoms is 1048576).

3 Challenges

The contribution presented here is not an exhaustive evaluation of Erlang/OTP systems. It is instead a collection of threats and potential mitigations that all Erlang developers and operators should be aware. Challenges can be grouped into three categories: Erlang/OTP level vulnerabilities, development and deployment related issues.

3.1 Erlang/OTP Level Vulnerabilities

Many vulnerabilities that exist on the Erlang/OTP level are due to its original design.

Erlang/OTP distribution protocol is by default insecure and none of the secure implementations supports transition (e.g., rolling upgrades). This can even mean a full outage since live Erlang clusters need to be taken down for maintenance in order to enable the more secure implementation. In case the developer chooses to use the built-in support for distribution over TLS, it is recommended to turn off the epmd since its implementation is still insecure, therefore, the secure option is quasi equivalent in terms of functionalities to the insecure one in some level.

The distribution protocol also makes Erlang nodes easily attackable if one knows the port the Erlang node listens to. Establishing a connection between two Erlang nodes starts with sending the name of the node who initiates the connection to the other node through TCP. The receiving part takes the name of the connecting node and turns it into an atom. A series of these connection requests, each with a different name, can then easily bring down the receiving node since its atom table will be filled.

Even though Erlang/OTP allows users to restrict the nodes that are allowed to connect to a node once the node is up and running by calling `net_kernel:allow/1` and passing the list of allowed nodes and hostnames, the implementation first turns the node name to an atom and only after then checks if the node is allowed to connect. Note that from the Erlang/OTP 21 release the implementation has been improved and the node name will only be turned into an atom once the `net_kernel:allow/1` check has succeeded. However, according to the official documentation, a list of nodes should be provided that limits the possibility of attaching to running Erlang nodes using remote Erlang shells.

We've seen that Erlang/OTP systems need to be hardened to stop attackers from bringing Erlang nodes down and from connecting to any Erlang node of the cluster. The reason why the later is of utmost importance is because there are no limits imposed by Erlang to what the attacker can do once connected. For instance, Erlang RPC allows the attacker to connect to all the other nodes of the cluster, even if they were not accessible to the attacker through the network in the first place. Also, using the `cmd` function of the `os` module the attacker can cause damages limited only by the Operating System level access control of the system user that runs the Erlang VM.

3.2 Development Related Issues

Erlang provides a wide range of libraries and the language itself supports rapid prototyping. Developers should, however, always keep in mind that (1) Erlang libraries may not be shipped with the most secure configuration enabled by default and (2) they need to protect the VM.

Establishing communication with a third-party system is just a few lines of code, however, Erlang's TLS library (`ssl` module) is not secure by default. All the releases prior to Erlang/OTP 21 have insecure cipher suites enabled by default, e.g., cipher suites that use Triple Data Encryption Algorithm (also known as 3DES) when used in Cipher Block Chaining (CBC) mode. The Erlang/OTP 21 release fixes this but these cipher suites have been considered weak [2] since 2016.

Moreover, how TLS handshake errors are handled by default is also insecure. Here we show that TLS handshake errors, for instance, the ones caused by revoked certificates, certificates signed by an untrusted root and expired certificates are ignored. In the particular case of Figure 1 the server is providing an expired certificate and still the TLS connection is established. Even though supplying the option `verify` with the value `verify_peer` and an up to date certificate store bundle, like in Figure 2, stops the establishment of the connection for the above mentioned TLS handshake errors, more parameters are required to make Erlang consider Certificate Revocation Lists, like is shown on Figure 3. What makes this even more challenging is that libraries that depend on the `ssl` module, delegate the responsibility of choosing the secure `ssl_options` to the developer. Examples of this are the `httpc` library or the community-driven `lhttpc`.

Last but not least, it is generally known that the ERTS has system limits. Exceeding a system limit will bring down the node, hence that should be actively guarded against.

[2] https://github.com/okeuday/pest

```
application:ensure_all_started(ssl),
ssl:connect("expired.badssl.com", 443, []).
```

Figure 1. Opening TLS connection towards host with expired server certificate.

```
application:ensure_all_started(ssl),
ssl:connect("untrusted-root.badssl.com",
            443,
            [ {verify, verify_peer}
            , {cacertfile, "bundle.pem"}
            ]).
```

Figure 2. Opening TLS connection towards host with server certificate signed by untrusted root.

```
application:ensure_all_started(ssl),
ssl:connect("untrusted-root.badssl.com",
            443,
            [ {verify, verify_peer}
            , {cacertfile, "bundle.pem"}
            , {crl_check, true}
            , {crl_cache, { ssl_crl_cache
                          , { internal
                            , [{http, 1000}]
                            }
                          }
              }
            ]).
```

Figure 3. Opening TLS connection towards host with revoked server certificate.

However, in day to day work, it is easy to forget about it and mitigation require a lot of boilerplate code.

3.3 Deployment Issues

A vast amount of open-sourced Erlang libraries has been released in the recent years. They are ready components that anyone can use and deploy. Consider Riak as an example. Deploying a Riak cluster requires a few minutes and the reward is great: a scalable, highly available key-value store is ready to use. However, it is easy to forget to change the cookie, which is `riak` by default. Cookies shall be changed, should be long and random enough not to be predictable. Moreover, the default configuration should be thoroughly reviewed and adjusted.

Since the Erlang distribution protocol is fragile, developers must harden the security by restricting the access to the distribution port of the Erlang nodes and restrict as much as possible what the system user that is running the Erlang VM can do to the system.

4 Techniques and Tools

The examples shown in Section 3 demonstrate that there is a lot of work ahead of us. The work falls into two categories: identifying, reporting and knowing about the vulnerabilities that exist in the Erlang eco-system and applying this knowledge to the running Erlang systems.

Running Erlang systems should be thoroughly reviewed. Each external library should be checked carefully, keeping only those that are from a trusted vendor and have been battle-tested. Vulnerabilities detected in libraries should be reported back to the community. Also, it is important to keep in mind that released software components may not be shipped with the most secure configuration by default, thus studying the configuration options that the used libraries provide is worth the effort.

However, manual reviews involving a human expert are always prone to errors. Thus, we need tools that can be used. The PEST project is a great first step towards this direction but we need more. We need precise tools that help performing security audits. Security audit should not be a one-time effort, instead, it should happen periodically. During the audit the source code *and* the production environment should be checked at least.

In the field of the static program analysis of Erlang programs significant research has already been carried out that can serve as a basis to audit source code [3, 6]. Detecting unsafe usage of user input could be significantly improved. Consider the `list_to_atom/1` calls as an example. Returning back all references of `list_to_atom/1` can make the result unusable for the developer since it includes safe usages as well. Besides, finding all references of this function is just the beginning of the work since the origin of their parameter needs to be detected and analysed. Here the really interesting thing is to find all entry points into the system where a user input is accepted but not checked and can flow unmodified into a `list_to_atom/1` call.

Environment checks are important to prevent human errors, which can be implemented as host scans. The scanner should check the network setup (e.g., firewalls, open ports), the host (e.g., what are the rights of the user running the Erlang VM) and the configuration of the Erlang system (e.g., has the default cookie been changed).

Last but not least, real-time monitoring the properties of the system can be used to detect anomalies or bad trends. For example, seeing a big spike in the average size of HTTP requests served by the system can be the sign of an attack.

5 Outlook

In this paper, we showed the type of challenges the Erlang community is facing and some techniques that can be used to tackle the challenges. This is just the beginning of the journey.

Security awareness is not a one time task. The system needs to be built with utmost care. Developers must consider the security aspects while doing rapid prototyping. Operators must periodically check for vulnerabilities. Hence, we need to build the security awareness in the community. And we need tools. Therefore, to secure all Erlang systems we need the collaboration of industry and academy – the whole Erlang community.

Acknowledgments

The authors would like to thank Dániel Szoboszlay for his feedback.

References

[1] Joe Armstrong. 2007. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 6–1–6–26. https://doi.org/10.1145/1238844.1238850

[2] Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 456–467. https://doi.org/10.1145/2976749.2978423

[3] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. 2008. Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG (ERLANG '08)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1411273.1411283

[4] Kenji Rikitake and Koji Nakao. 2008. Application Security of Erlang Concurrent Systems. In *Computer Security Symposium, CSS*, Vol. 8.

[5] Kenji Rikitake and Koji Nakao. 2009. SSH Distribution Transport on Erlang Concurrent System. In *Proceedings of IPSJ Computer Security Symposium*. 117–122.

[6] Melinda Tóth and István Bozó. 2012. Static Analysis of Complex Software Systems Implemented in Erlang. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School (CEFP'11)*. Springer-Verlag, Berlin, Heidelberg, 440–498. https://doi.org/10.1007/978-3-642-32096-5_9